



Compiler-Based Performance Autotuning Opportunities

Boyana Norris, norris@mcs.anl.gov

Computer Scientist, Argonne National Laboratory

<http://www.mcs.anl.gov/~norris>

Representing the **Performance Engineering Research Institute (PERI)** SciDAC Project

<http://www.peri-scidac.org>

PERI: Performance Engineering Research Institute

- Performance measurement
 - Profiling, tracing
 - Performance data management
- Performance analysis
 - Profile and trace-based
 - Source-based
 - Manual
- Performance optimization
 - Automated
 - Manual



Performance tools (also see <http://peri-scidac.org>)

- ActiveHarmony (runtime adaptation, empirical tuning)
 - <http://www.dyninst.org/harmony/>
- CHILL (loop transformations)
 - <http://www.chunchen.info/chill/>
- HPCToolkit (profiling, analysis)
 - <http://hpctoolkit.org>
- Orio (annotations, transformations, empirical tuning)
 - <http://tinyurl.com/OrioTool>
- PAPI (measurement)
 - <http://icl.cs.utk.edu/papi/>
- ROSE (compiler construction infrastructure)
 - <http://rosecompiler.org>
- TAU (profiling, tracing, analysis)
 - <http://www.cs.uoregon.edu/research/tau>





This talk: **Compiler-Based Performance Autotuning**



Motivation: A looming software crisis

- Architectures are getting increasingly complex
 - Multiple cores, deep memory hierarchies, software-controlled storage, shared resources, SIMD compute engines, heterogeneity, ...
- Performance optimization is getting more important
 - Today's sequential and parallel applications **may not** be faster on tomorrow's architectures.
 - Especially if you want to add new capability!
 - Managing **data locality** even more important than parallelism.
 - Managing **power** of growing importance, too.
- Performance portability
 - Tuning for a particular architecture potentially hinders performance on different architectures

Complexity!

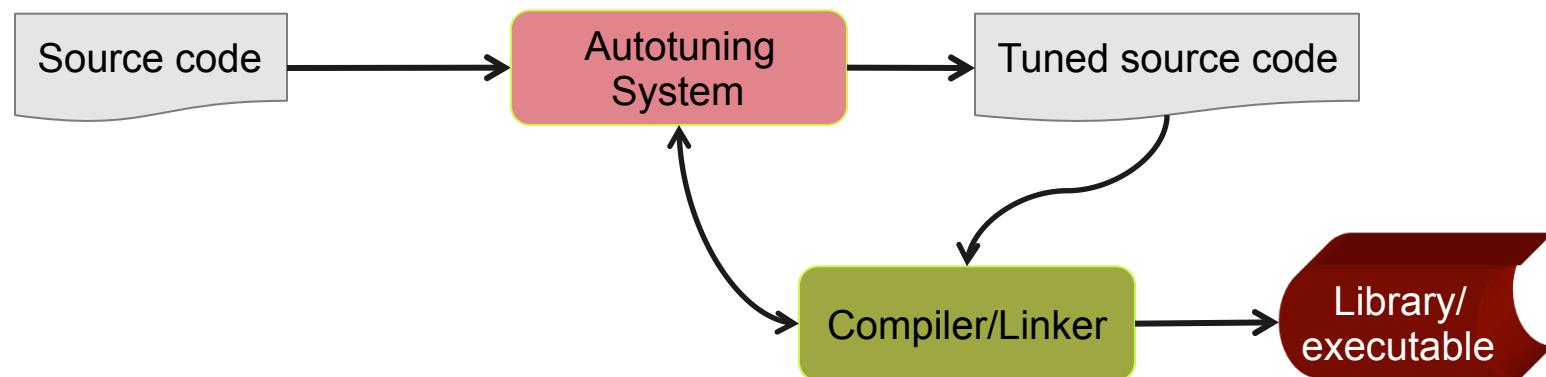


What is autotuning?

- Goal: Maximize achieved performance
- Traditional compilation:



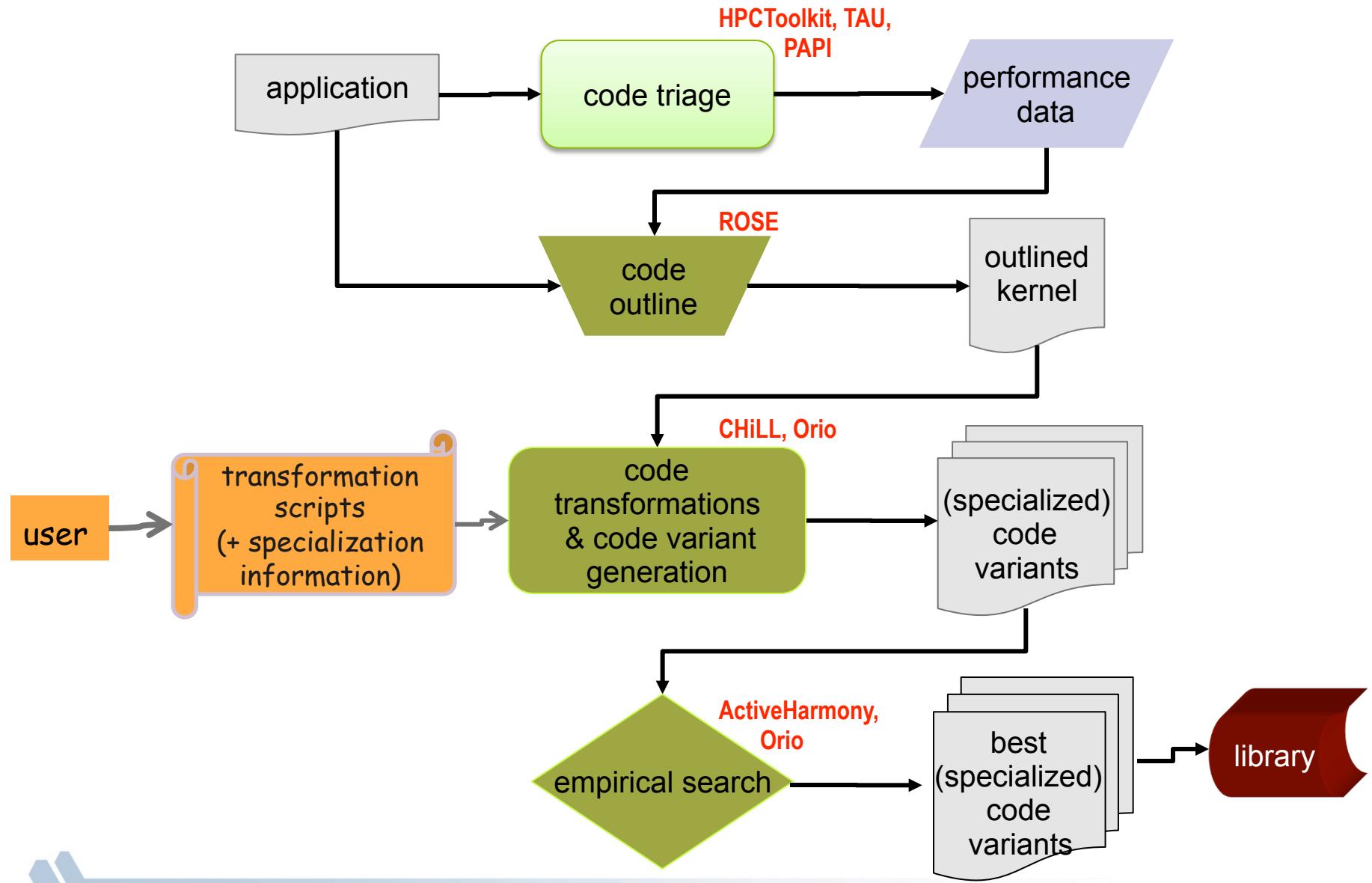
- Autotuning:



- *Some compilers (e.g., Cray) employ autotuning internally*



Example of an autotuning process in PERI



Requirements for compiler-based autotuning

- Parsing infrastructure
- Program analysis (interprocedural dataflow, alias, etc.)
- Source code generation
- Empirical search

- Some overlaps with other DOE research areas: reuse of analysis in other source-to-source systems, e.g.,
 - Automatic differentiation
 - Source-based performance analysis



Transformations and parameters (some examples)

Transformations	Definition	Goal	Variants	Parameters
Loop permutation	Change the loop order	Enable U&J and Tiling + Reduce TLB misses	Different loop orders	-
Unroll and Jam	Unroll outer loops and fuse inner loops			
Scalar replacement	Replace array accesses with scalar variables	Reuse in registers	-	Unroll factors
Tiling	Divide iteration space into tiles	cache utilization; multicore parallelization	-	Tile sizes
Data copying (w/ tiling)	Copy subarray into contiguous memory space	Avoid conflict misses + Avoid TLB thrashing	Yes/no on specific data structures	-
Prefetching	Prefetch data into cache before actual references	Hide memory latency	-	Prefetch distances

- ★ All loops are unrolled and tiled and all data are prefetched.
- ★ For degenerate cases, Unroll factor=1, Tile size=1 and Prefetch distance=0, code transformations are not applied.



(Semi-)Automated empirical performance tuning

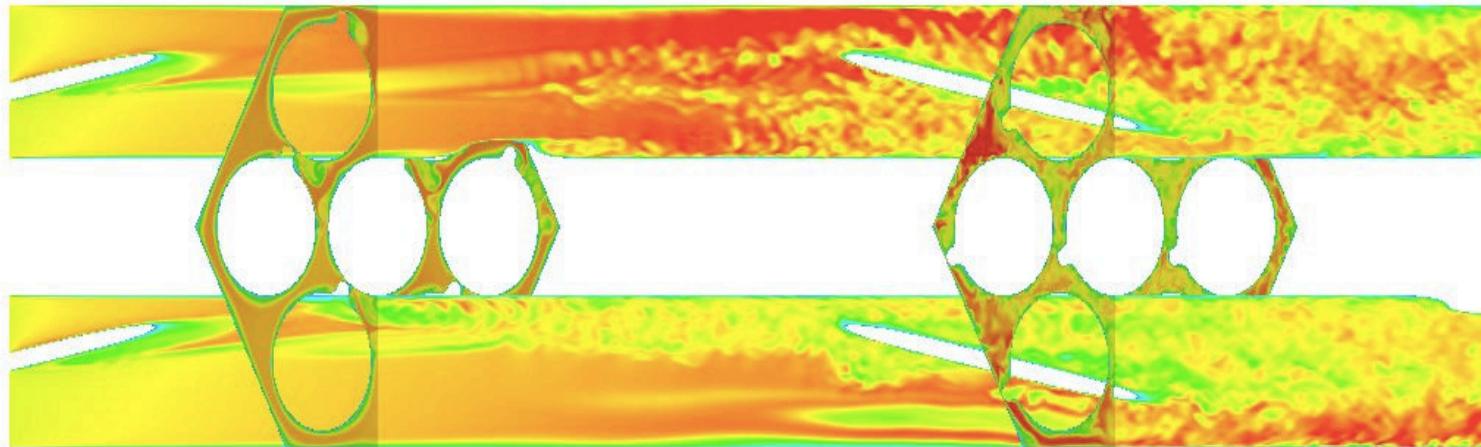
□ Problems:

- Large number of parameters to tune
- Shape of objective function unknown
- Multiple libraries and coupled applications
- Analytical model may not be available



Example: Autotuning of Nek5000

Spectral element code: turbulence in wire-wrapped subassemblies



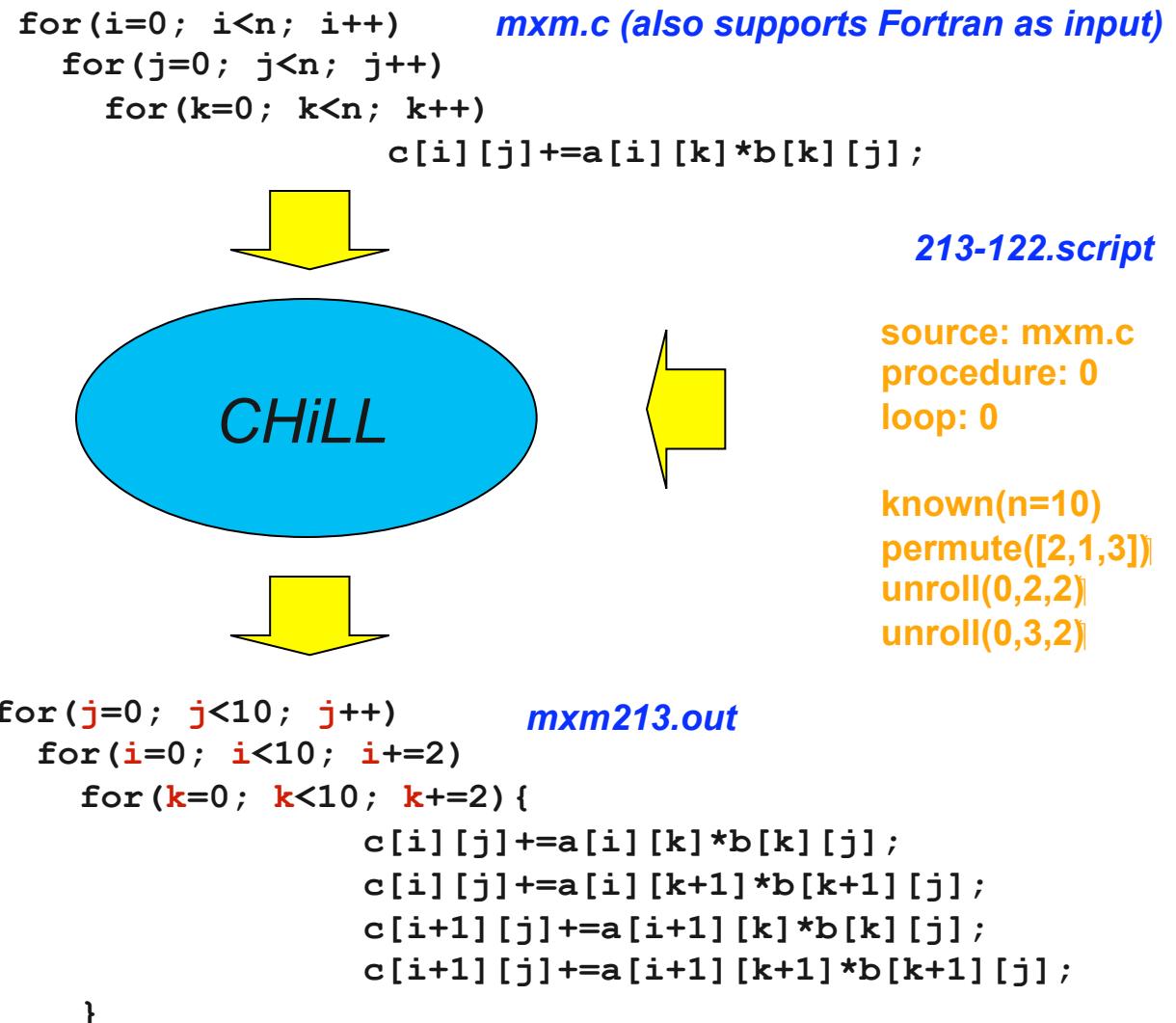
- Applications: nuclear energy, astrophysics, ocean modeling, combustion, bio fluids,
- Scales to $P > 10,000$ (Cray XT5, BG/P)
- > 75% of time spent on manually optimized mxm
 - matrix multiply of very small, rectangular matrices
 - matrix sizes remain the same for different problem sizes



Example (cont.): Generate library automatically using CHiLL

High-level loop transformation and code generation framework

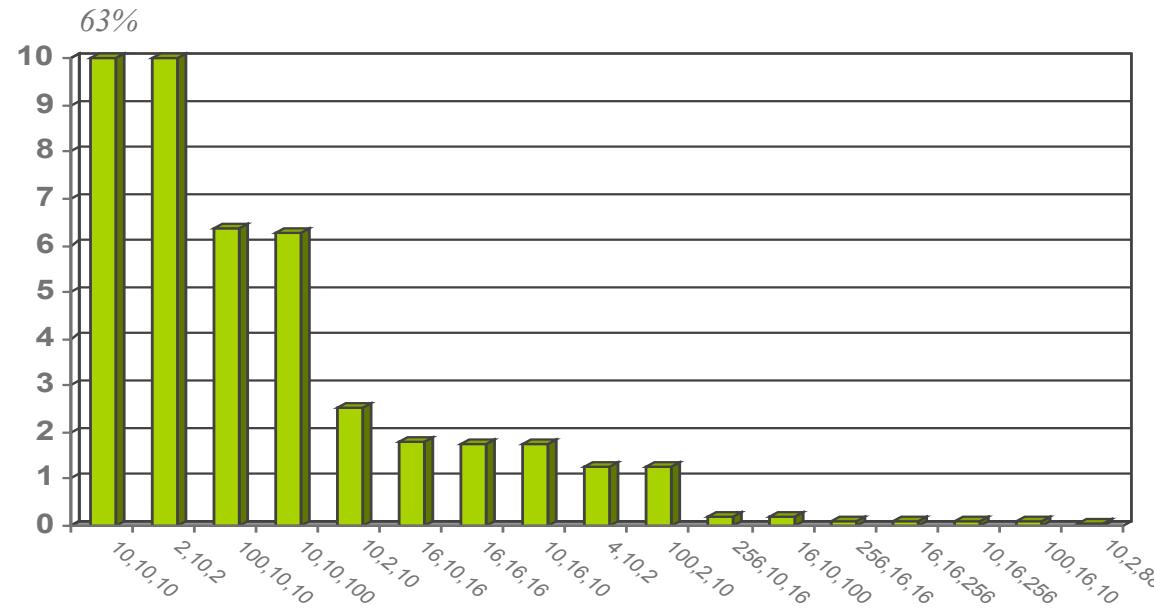
- based on polyhedral model
- script interface for programmers or compilers
- optimization strategy expressed as sequence of composable transformations



Source: Mary Hall

Example (cont.): BLAS library is significant to performance

- 8 input sizes comprise 75% of time
- Optimization opportunities
 - exploit reuse in registers (**unroll-and-jam**)
 - exploit SIMD (in the Opteron SSE-3) (**permute, unroll**)
 - reduce loop overheads (**unroll, specialize**)



Example (cont.):

Generated code: Would anyone want to write this?

Example: loop order ijk, unroll 8-4-1

FUNCTION M_100_10_8 (A, B, C)

```
INTEGER M_100_10_8, T4, T6
DOUBLE PRECISION A, B, C

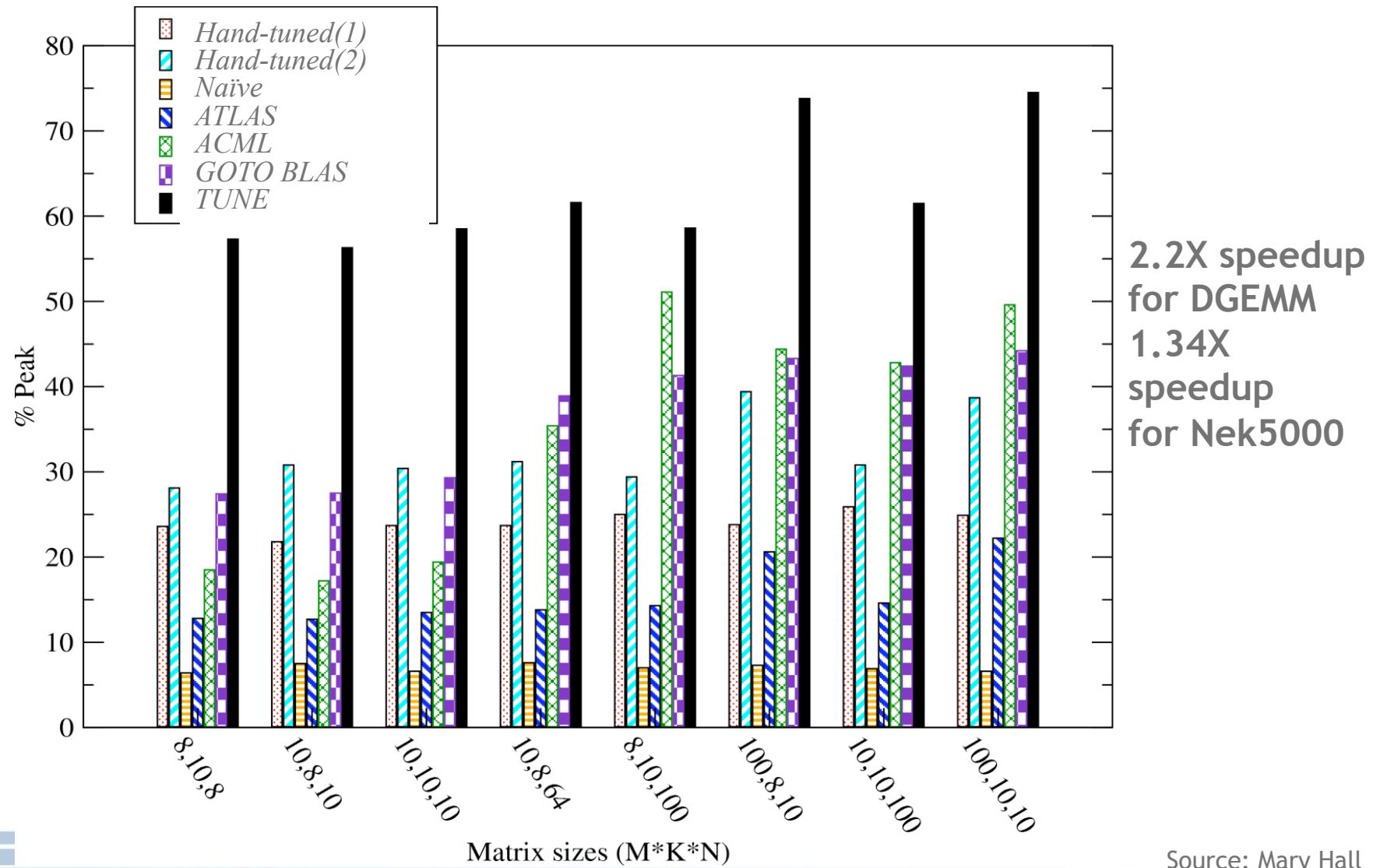
DIMENSION A(8, 10)
DIMENSION B(10, 100)
DIMENSION C(8, 100)

DO 2, T4 = 1, 97, 4
C(1, T4) = 0.0000000000000000D+00
C(1 + 1, T4) = 0.0000000000000000D+00
C(1 + 2, T4) = 0.0000000000000000D+00
C(1 + 3, T4) = 0.0000000000000000D+00
C(1 + 4, T4) = 0.0000000000000000D+00
C(1 + 5, T4) = 0.0000000000000000D+00
C(1 + 6, T4) = 0.0000000000000000D+00
C(1 + 7, T4) = 0.0000000000000000D+00
C(1, T4 + 1) = 0.0000000000000000D+00
C(1 + 1, T4 + 1) = 0.0000000000000000D+00
C(1 + 2, T4 + 1) = 0.0000000000000000D+00
C(1 + 3, T4 + 1) = 0.0000000000000000D+00
C(1 + 4, T4 + 1) = 0.0000000000000000D+00
C(1 + 5, T4 + 1) = 0.0000000000000000D+00
C(1 + 6, T4 + 1) = 0.0000000000000000D+00
C(1 + 7, T4 + 1) = 0.0000000000000000D+00
C(1, T4 + 2) = 0.0000000000000000D+00
C(1 + 1, T4 + 2) = 0.0000000000000000D+00
C(1 + 2, T4 + 2) = 0.0000000000000000D+00
C(1 + 3, T4 + 2) = 0.0000000000000000D+00
C(1 + 4, T4 + 2) = 0.0000000000000000D+00
C(1 + 5, T4 + 2) = 0.0000000000000000D+00
C(1 + 6, T4 + 2) = 0.0000000000000000D+00
C(1 + 7, T4 + 2) = 0.0000000000000000D+00
C(1, T4 + 3) = 0.0000000000000000D+00
C(1 + 1, T4 + 3) = 0.0000000000000000D+00
C(1 + 2, T4 + 3) = 0.0000000000000000D+00
C(1 + 3, T4 + 3) = 0.0000000000000000D+00
C(1 + 4, T4 + 3) = 0.0000000000000000D+00
C(1 + 5, T4 + 3) = 0.0000000000000000D+00
C(1 + 6, T4 + 3) = 0.0000000000000000D+00
C(1 + 7, T4 + 3) = 0.0000000000000000D+00
```

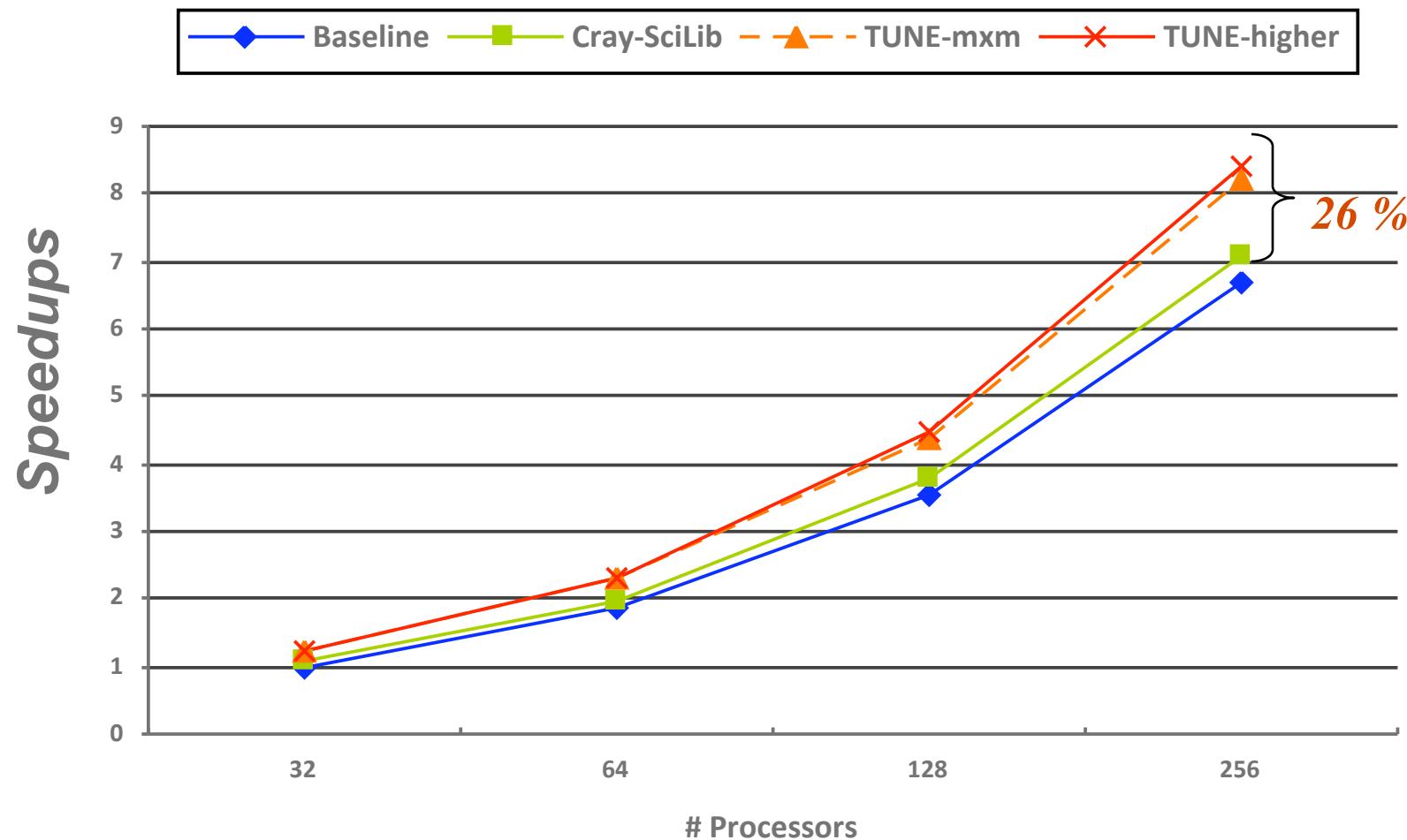
```
DO 4, T6 = 1, 10, 1
C(1, T4) = C(1, T4) + A(1, T6) * B(T6, T4)
C(1 + 1, T4) = C(1 + 1, T4) + A(1 + 1, T6) * B(T6, T4)
C(1 + 2, T4) = C(1 + 2, T4) + A(1 + 2, T6) * B(T6, T4)
C(1 + 3, T4) = C(1 + 3, T4) + A(1 + 3, T6) * B(T6, T4)
C(1 + 4, T4) = C(1 + 4, T4) + A(1 + 4, T6) * B(T6, T4)
C(1 + 5, T4) = C(1 + 5, T4) + A(1 + 5, T6) * B(T6, T4)
C(1 + 6, T4) = C(1 + 6, T4) + A(1 + 6, T6) * B(T6, T4)
C(1 + 7, T4) = C(1 + 7, T4) + A(1 + 7, T6) * B(T6, T4)
C(1, T4 + 1) = C(1, T4 + 1) + A(1, T6) * B(T6, T4 + 1)
C(1 + 1, T4 + 1) = C(1 + 1, T4 + 1) + A(1 + 1, T6) * B(T6, T4 + 1)
C(1 + 2, T4 + 1) = C(1 + 2, T4 + 1) + A(1 + 2, T6) * B(T6, T4 + 1)
C(1 + 3, T4 + 1) = C(1 + 3, T4 + 1) + A(1 + 3, T6) * B(T6, T4 + 1)
C(1 + 4, T4 + 1) = C(1 + 4, T4 + 1) + A(1 + 4, T6) * B(T6, T4 + 1)
C(1 + 5, T4 + 1) = C(1 + 5, T4 + 1) + A(1 + 5, T6) * B(T6, T4 + 1)
C(1 + 6, T4 + 1) = C(1 + 6, T4 + 1) + A(1 + 6, T6) * B(T6, T4 + 1)
C(1 + 7, T4 + 1) = C(1 + 7, T4 + 1) + A(1 + 7, T6) * B(T6, T4 + 1)
C(1, T4 + 2) = C(1, T4 + 2) + A(1, T6) * B(T6, T4 + 2)
C(1 + 1, T4 + 2) = C(1 + 1, T4 + 2) + A(1 + 1, T6) * B(T6, T4 + 2)
C(1 + 2, T4 + 2) = C(1 + 2, T4 + 2) + A(1 + 2, T6) * B(T6, T4 + 2)
C(1 + 3, T4 + 2) = C(1 + 3, T4 + 2) + A(1 + 3, T6) * B(T6, T4 + 2)
C(1 + 4, T4 + 2) = C(1 + 4, T4 + 2) + A(1 + 4, T6) * B(T6, T4 + 2)
C(1 + 5, T4 + 2) = C(1 + 5, T4 + 2) + A(1 + 5, T6) * B(T6, T4 + 2)
C(1 + 6, T4 + 2) = C(1 + 6, T4 + 2) + A(1 + 6, T6) * B(T6, T4 + 2)
C(1 + 7, T4 + 2) = C(1 + 7, T4 + 2) + A(1 + 7, T6) * B(T6, T4 + 2)
C(1, T4 + 3) = C(1, T4 + 3) + A(1, T6) * B(T6, T4 + 3)
C(1 + 1, T4 + 3) = C(1 + 1, T4 + 3) + A(1 + 1, T6) * B(T6, T4 + 3)
C(1 + 2, T4 + 3) = C(1 + 2, T4 + 3) + A(1 + 2, T6) * B(T6, T4 + 3)
C(1 + 3, T4 + 3) = C(1 + 3, T4 + 3) + A(1 + 3, T6) * B(T6, T4 + 3)
C(1 + 4, T4 + 3) = C(1 + 4, T4 + 3) + A(1 + 4, T6) * B(T6, T4 + 3)
C(1 + 5, T4 + 3) = C(1 + 5, T4 + 3) + A(1 + 5, T6) * B(T6, T4 + 3)
C(1 + 6, T4 + 3) = C(1 + 6, T4 + 3) + A(1 + 6, T6) * B(T6, T4 + 3)
C(1 + 7, T4 + 3) = C(1 + 7, T4 + 3) + A(1 + 7, T6) * B(T6, T4 + 3)
4 CONTINUE
3 CONTINUE
2 CONTINUE
1 CONTINUE
M_100_10_8 = 0
RETURN
END
```



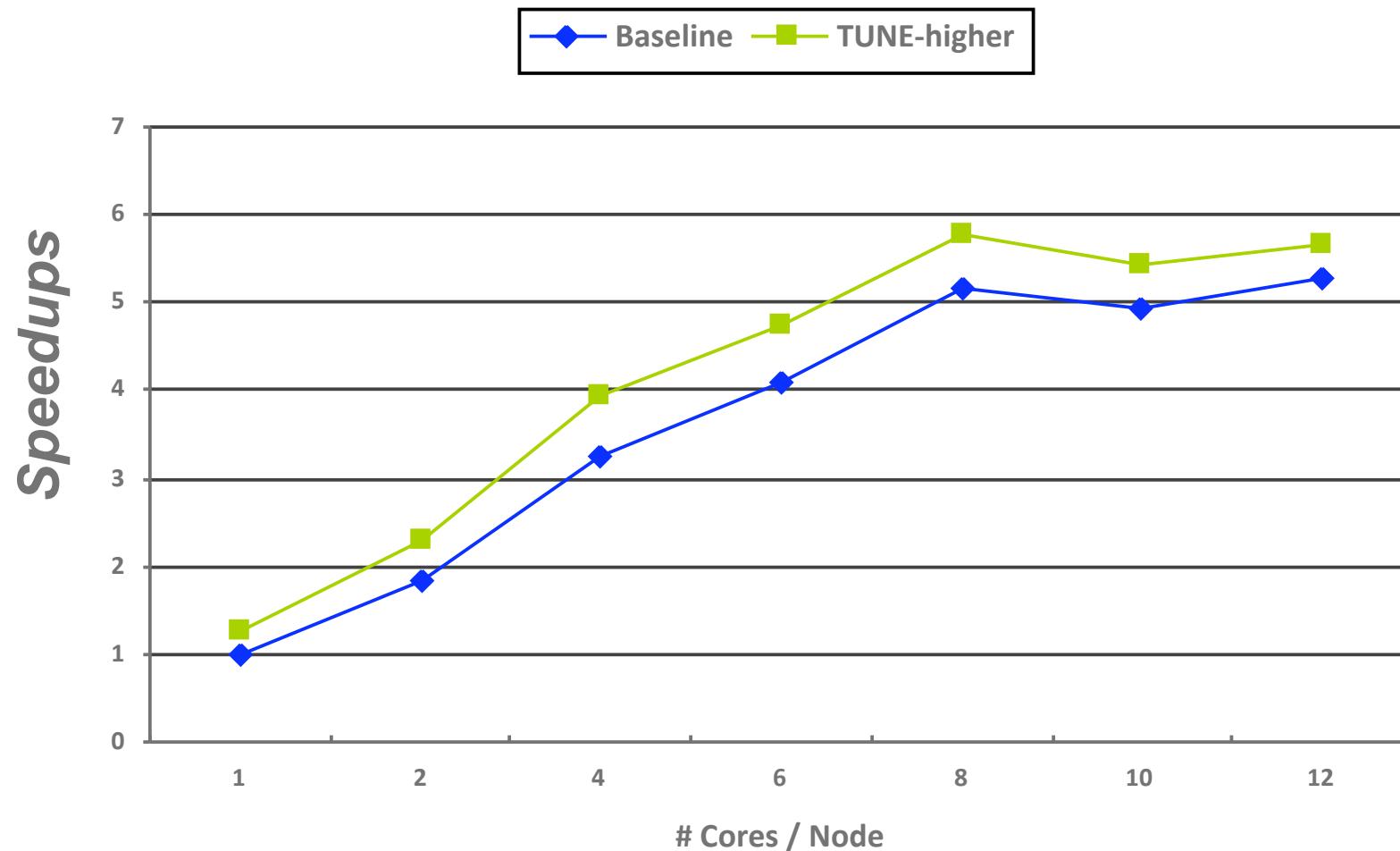
Automatically generated code is faster than manually-tuned libraries



Example (cont.): Nek5000 (g6a input) on Jaguar



Example (cont.): Multiple Cores / Node: Nek5000 (g6a) on 32 nodes of Jaguar



Current capabilities

- Semi-automated tuning of kernels (focus on single-node performance)
 - Manual:
 - Transformation (and in some cases input) specifications
 - Partly automated:
 - Triage
 - Kernel extraction
 - Fully automated:
 - Transformations
 - Empirical search



Future directions

- SBIR relevant topics: making tools robust, better automation, user interfaces
- Some examples:
 - More automation, e.g. in triage, transformation specification
 - More extensive use of performance models
 - Advances in empirical search: multiobjective derivative-free optimization approaches
 - Tuning of larger subsets of functionality (not just small-ish kernels)
 - Tuning for heterogeneous architectures
 - Beyond performance: reliability, resilience
- Possible SBIR topics in areas that share infrastructure with tools involved in autotuning:
 - Automatic differentiation
 - Source-based performance analysis



Performance tools (also see <http://peri-scidac.org>)

- ActiveHarmony (runtime adaptation, empirical tuning)
 - <http://www.dyninst.org/harmony/>
- CHILL (loop transformations)
 - <http://www.chunchen.info/chill/>
- HPCToolkit (profiling, analysis)
 - <http://hpctoolkit.org>
- Orio (annotations, transformations, empirical tuning)
 - <http://tinyurl.com/OrioTool>
- PAPI (measurement)
 - <http://icl.cs.utk.edu/papi/>
- ROSE (compiler construction infrastructure)
 - <http://rosecompiler.org>
- TAU (profiling, tracing, analysis)
 - <http://www.cs.uoregon.edu/research/tau>

